



Maestría en Ciencias de la Computación

Algoritmos y Estructura de Datos (AyED)

Práctica 01: Algoritmos de Ordenamiento

Docente: Dr. Vicente Machaca Arceda

Integrantes:

Franklin Canaza Ccori
Hayde Luzmila Humpire Cutipa
Jair Francesco Huaman Canqui
Jhoel Salomon Tapara Quispe

Universidad Nacional de San Agustín de Arequipa
Agosto de 2022

Índice

1. Introducción	8
1.1. Requerimientos	8
1.2. Generación de Datos	8
2. Algoritmos	9
2.1. <i>Counting Sort</i>	9
2.2. <i>Quick Sort</i>	10
2.3. <i>Insertion Sort</i>	12
2.4. <i>Heap Sort</i>	13
3. Implementación	14
3.1. <i>Counting Sort</i>	14
3.1.1. C++	14
3.1.2. Python	15
3.1.3. Goland	15
3.2. <i>Quick Sort</i>	16
3.2.1. C++	16
3.2.2. Python	17
3.2.3. Goland	17
3.3. <i>Insertion Sort</i>	18
3.3.1. C++	18
3.3.2. Python	18
3.3.3. Goland	19
3.4. <i>Heap Sort</i>	19
3.4.1. C++	19
3.4.2. Python	20

3.4.3.	Goland	20
4.	Resultados	21
4.1.	Comparación con Python, C++ y Golang	22
4.1.1.	Comparación de <i>Counting Sort</i>	22
4.1.2.	Comparación de <i>Quick Sort</i>	22
4.1.3.	Comparación de <i>Insertion Sort</i>	23
4.1.4.	Comparación <i>Heap Sort</i>	23
4.2.	Comparación con C++	24
4.2.1.	Comparación de Algoritmos en C++	24
4.2.2.	<i>Insertion Sort</i> en C++	25
4.2.3.	<i>Counting Sort</i> en C++	25
4.2.4.	<i>Heap Sort</i> en C++	26
4.2.5.	Quick <i>Quick Sort</i> en C++	26
4.3.	Comparación con Golang	27
4.3.1.	Comparación de Algoritmos en Golang	27
4.3.2.	<i>Insertion Sort</i> en Golang	27
4.3.3.	<i>Counting Sort</i> en Golang	28
4.3.4.	<i>Heap Sort</i> en Golang	28
4.3.5.	<i>Quick Sort</i> en Golang	29
4.4.	Comparación con Python	30
4.4.1.	Comparación de Algoritmos en Python	30
4.4.2.	<i>Insertion Sort</i> en Python	30
4.4.3.	<i>Counting Sort</i> en Python	31
4.4.4.	<i>Heap Sort</i> en Python	31
4.4.5.	<i>Quick Sort</i> en Python	32
4.5.	Tablas Comparativa de Tiempos	32

4.5.1.	Tiempos de ejecución: <i>Counting Sort</i>	33
4.5.2.	Tiempos de ejecución: <i>Quick Sort</i>	33
4.5.3.	Tiempos de ejecución: <i>Insertion Sort</i>	34
4.5.4.	Tiempos de ejecución: <i>Heap Sort</i>	34
5.	Conclusiones	35

Índice de figuras

1.	Archivo Puntos.txt	9
2.	Ejemplo: <i>Counting Sort</i>	10
3.	Ejemplo: <i>Quick Sort</i>	11
4.	Ejemplo: <i>Insertion Sort</i>	12
5.	Ejemplo: <i>Max Heap</i>	13
6.	Ejemplo: <i>Min Heap</i>	13
7.	Comparación de <i>Counting Sort</i>	22
8.	Comparación de <i>Quick Sort</i>	23
9.	Comparación de <i>Insertion Sort</i>	23
10.	Comparación de <i>Heap Sort</i>	24
11.	Comparación de Algoritmos en C++	24
12.	Comparación de <i>Insertion Sort</i>	25
13.	Comparación de <i>Counting Sort</i>	25
14.	Comparación de <i>Heap Sort</i>	26
15.	Comparación de <i>Quick Quick Sort</i>	26
16.	Comparación de Algoritmos en Golang	27
17.	<i>Insertion Sort</i> en Golang	28
18.	<i>Counting Sort</i> en Golang	28
19.	<i>Heap Sort</i> en Golang	29
20.	<i>Quick Sort</i> en Golang	29
21.	Comparación de Algoritmos en Python	30
22.	<i>Insertion Sort</i> en Python	31
23.	<i>Counting Sort</i> en Python	31
24.	<i>Heap Sort</i> en Python	32

25. <i>Quick Sort</i> en Python	32
-------------------------------------------	----

Índice de cuadros

1.	Complejidad: <i>Counting Sort</i>	10
2.	Complejidad: <i>Quick Sort</i>	10
3.	Complejidad: <i>Insertion Sort</i>	12
4.	Complejidad: <i>Heap Sort</i>	13
5.	Tiempos de ejecución del algoritmo <i>Counting Sort</i>	33
6.	Tiempos de ejecución del algoritmo <i>Quick Sort</i>	33
7.	Tiempos de ejecución del algoritmo <i>Insertion Sort</i>	34
8.	Tiempos de ejecución del algoritmo <i>Heap Sort</i>	34

1. Introducción

Un algoritmo de ordenamiento se encarga de ubicar correctamente los elementos de un vector, ya sea en un orden ascendente o descendente. Los algoritmos de ordenamiento hacen uso de diversas técnicas, para lo cual un algoritmo de ordenamiento es eficiente en términos de tiempo y memoria, es por ello que se busca comparar sus resultados en el procesamiento de arreglos de bastantes elementos.

1.1. Requerimientos

Para realizar la comparación se han considerado los siguientes puntos:

1. Se han tomado 5 arreglos para determinar 5 mediciones de la velocidad a una determinada cantidad de elementos, de esta manera calcular la velocidad promedio y las desviación estándar para calcular los límites superior o inferior.
2. Se han generado arreglos con elementos aleatorios con dimensiones de 100, 1000, 2000, 3000, 4000, 5000, 6000, 7000, 8000, 9000, 10000, 20000, 30000, 40000 y 50000.

1.2. Generación de Datos

Para la generación de datos en archivos .txt se creó un script de nombre: generador.cpp con el lenguaje de programación C++, el cual genera números aleatorios y va escribiendo los mismo en un txt, genera 5 vectores de cada longitud. A continuación se muestra el código.

```
1 int main(){
2
3     //Parametros del numero de puntos
4     int NUM_ELEMENTOS=5;
5     int valores
6     []={100,1000,2000,3000,4000,5000,6000,7000,8000,9000,10000,20000,
7     30000,40000,50000};
8     int nro_puntos=sizeof(valores)/sizeof(valores[0]);
9
10    srand(time(NULL));
11    ofstream f("Puntos.txt");
12    string auxiliar;
13    int maximo;
14
15    f<<NUM_ELEMENTOS<<endl;
16
17    for(int i=0;i<nro_puntos;i++){
18        f<<valores[i]<<endl;
19        maximo=valores[i];
20        for(int j=0;j<NUM_ELEMENTOS;j++){
```

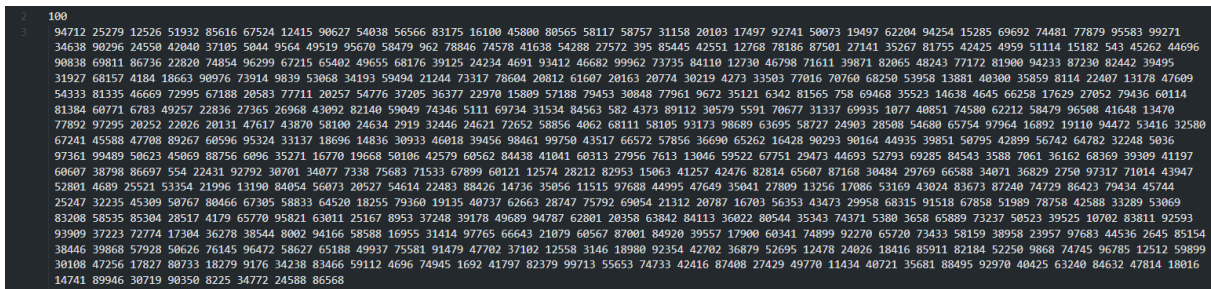


```

21 //Crear vector
22 auxiliar=" ";
23 for(int k=0;k<valores[i];k++){
24     auxiliar=auxiliar+ " " + to_string(1 + rand() % (maximo - 1));
25 }
26 f<<auxiliar<<endl;
27 }
28 }
29 f.close();
30 return 0;
31 }

```

Listing 1: Generación del archivo Puntos.txt



The image shows a screenshot of a text file named 'Puntos.txt'. The file contains a single line of approximately 1000 random integers, ranging from 14741 to 99271. The numbers are separated by spaces and wrap around the line. The first few numbers are 14741, 89946, 30719, 90350, 8225, 34772, 24588, 86568, 94712, 25279, 12526, 51932, 85616, 67524, 12415, 90627, 54038, 56566, 83175, 16100, 45800, 80565, 58117, 58757, 31158, 20103, 17497, 92741, 50073, 19497, 62204, 94254, 15285, 69692, 74481, 77879, 95583, 99271.

Figura 1: Archivo Puntos.txt

2. Algoritmos

Se han implementado 4 algoritmos, los cuales tienen las siguientes complejidades en tiempo:

1. *Counting Sort* (Conteo): Complejidad $O(n + k)$
2. *Quick Sort* (Rápido): Complejidad $O(n \log n)$
3. *Insertion Sort* (Inserción): Complejidad $O(n^2)$
4. *Heap Sort* (Montones): Complejidad $O(n \log n)$

2.1. *Counting Sort*

El algoritmo de ordenamiento *Counting Sort* consiste en contar la repetición de los elementos de un *array* ordenado, el cual contiene los elementos del mínimo al máximo es por ello que se crea un arreglo vacío con espacio entre el valor mínimo y máximo, para ir contando las repeticiones y luego indexarlas en un arreglo de salida.

Counting Sort, utiliza tres *arrays*: *Input*, *Output* y *Count*. Gracias a los índices de los *arrays*, el algoritmo puede colocar los valores reordenados directamente en el vector salida, todo el proceso tiene una complejidad en tiempo de:

<i>Counting Sort</i>	Mejor Caso	Caso Promedio	Peor Caso
Complejidad	$O(n+k)$	$O(n+k)$	$O(n+k)$

Cuadro 1: Complejidad: *Counting Sort*

Como se puede ver el ordenamiento es del tipo lineal en tiempo, sin embargo si los datos están muy esparsos, se puede llegar a separar demasiada memoria que cuente con valores 0, ahí radica una de sus debilidades.

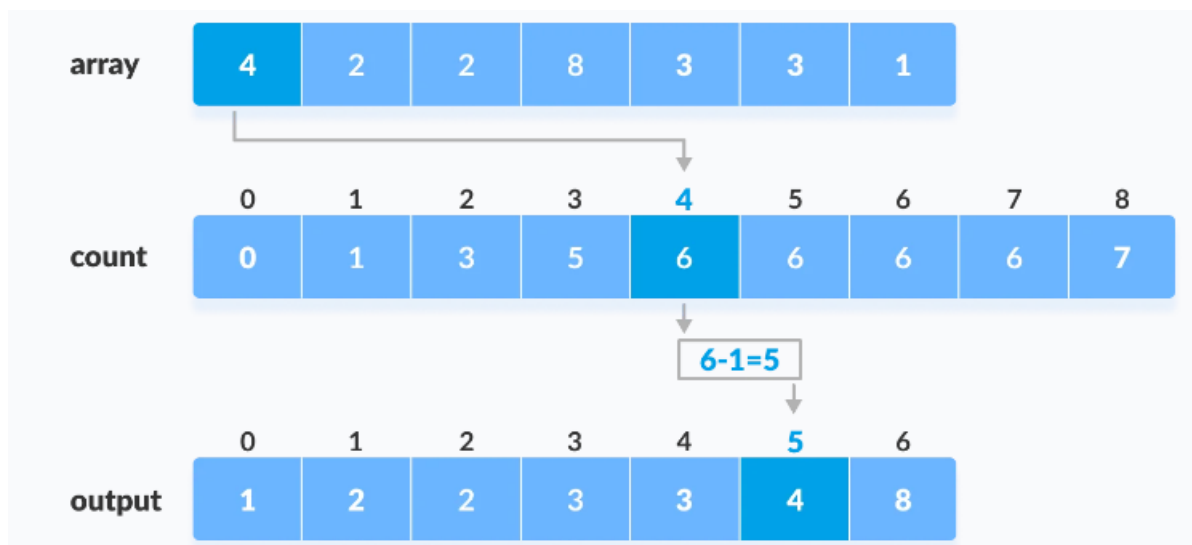


Figura 2: Ejemplo: *Counting Sort*

2.2. *Quick Sort*

El algoritmo *Quick Sort* se basa en la técnica de "divide y vencerás", dado que en cada recursión, el problema se divide en subproblemas de menor tamaño y se resuelven por separado (aplicando la misma técnica) para ser unidos de nuevo una vez resueltos, en base a ello su complejidad en el tiempo es de:

<i>Quick Sort</i>	Mejor Caso	Caso Promedio	Peor Caso
Complejidad	$O(n \log n)$	$O(n \log n)$	$O(n^2)$

Cuadro 2: Complejidad: *Quick Sort*.

El proceso es sencillo, consiste en asignar un pivote y ordenar cada elemento al lado izquierdo o derecho, según sean estos menores o mayores al pivote, luego aplicar de manera recursiva esta división con pivote a cada uno de los lados izquierdo y derecho, de esta

manera obtener los sub arreglos ordenados.

Un ejemplo de esto podemos verlo en la figura 3, donde se puede ver la representación en un árbol de esta división, en este caso se toma como pivote el ultimo valor, para crear los arreglos con elementos menores a 3 y mayores a 3, luego aplicar de manera recursiva la misma idea, formando en la izquierda los arreglos menores a 1 y a la derecha los arreglos mayores a 1, reproduciendo este procedimiento con los arreglos siguientes, es en base a este procedimiento que se logra un caso promedio de $n \log n$.

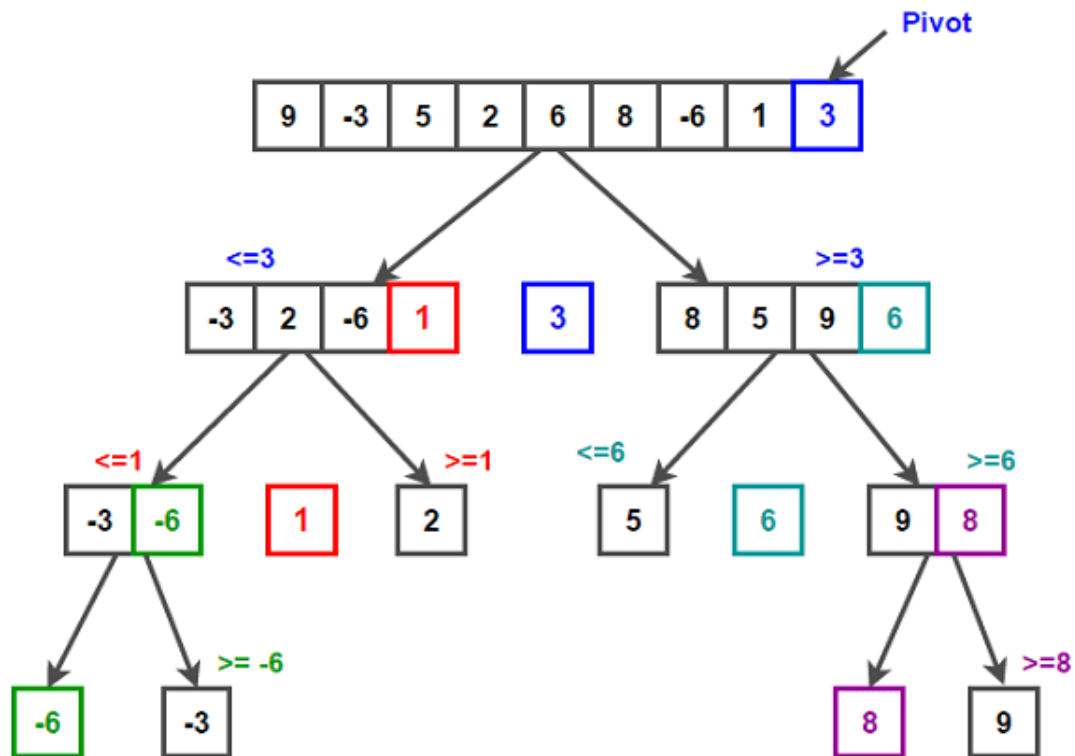


Figura 3: Ejemplo: *Quick Sort*

2.3. Insertion Sort

El algoritmo *Insertion Sort* funciona retirando un elemento de la lista de elementos y volviéndolo a poner en la lista pero esta vez en su posición correcta con respecto a una lista que ya se encuentra ordenada, para ello consigue los siguientes costes computacionales.

Insertion Sort	Mejor Caso	Caso Promedio	Peor Caso
Complejidad	$O(n)$	$O(n^2)$	$O(n^2)$

Cuadro 3: Complejidad: *Insertion Sort*

Para la aplicación de esta algoritmo se toma un elemento y se va comparando el mismo con sus elementos anteriores, hasta ubicarlo en su posición correcta, luego se procede con el siguiente elemento y así hasta ubicar correctamente todos los elementos del vector, algunas de sus desventajas es que realiza demasiadas comparaciones innecesarias, es por ello que logra ser de coste n^2 en el caso promedio un ejemplo de este procedimiento se puede ver en la imagen 4.

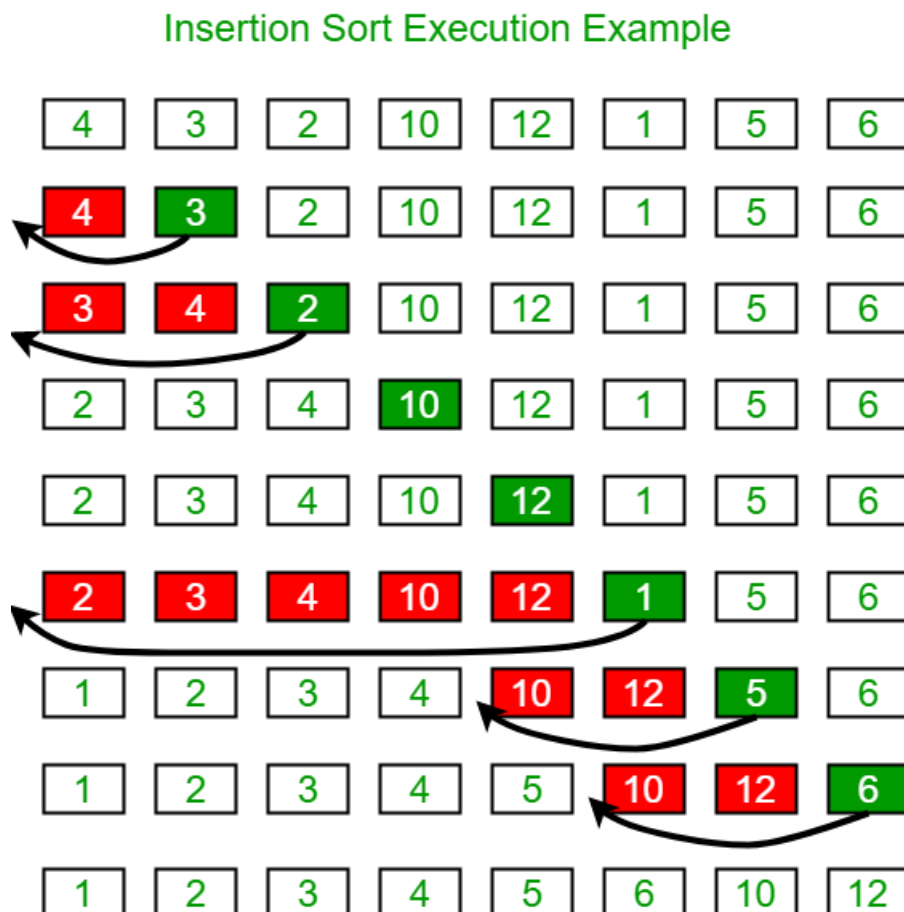


Figura 4: Ejemplo: *Insertion Sort*

2.4. *Heap Sort*

El algoritmo *Heap Sort* consiste en almacenar todos los elementos del vector a ordenar en un montículo (*heap*), y luego extraer el nodo que queda como nodo raíz del montículo (cima) en sucesivas iteraciones obteniendo el conjunto ordenado.

Heap Sort	Mejor Caso	Caso Promedio	Peor Caso
Complejidad	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$

Cuadro 4: Complejidad: *Heap Sort*.

El proceso consiste en tomar todo el arreglo y formar un árbol apartir de el, este mismo es sometido a un proceso *heapify*, el cual deja al elemento mayor en la raíz, para luego ser colocado en el tope del arreglo, para luego aplicar el *heapify* al resto del arreglo, así de manera recursiva hasta ordenar todo el arreglo.

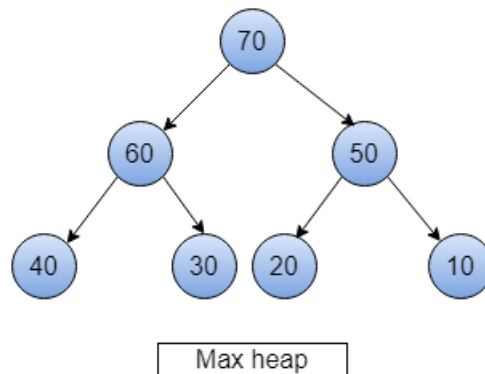


Figura 5: Ejemplo: *Max Heap*

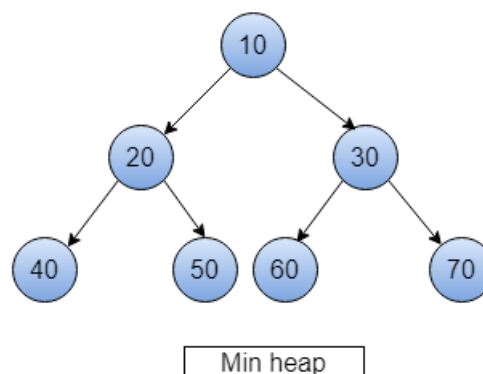


Figura 6: Ejemplo: *Min Heap*

Se puede usar el *heapify* para ordenar de manera ascendente o descendente, para ello se tiene las variantes de las figuras 5 y la figura 6.

3. Implementación

La implementación de la práctica se encuentra en el siguiente repositorio:
<https://github.com/HaydeLuzmilaHc/Grupo-01-Practicas-de-AyED>

Se va a presentar cada uno de los *scripts*, además de una breve discusión de cada uno de los algoritmos.

3.1. *Counting Sort*

Para poder aplicar el *counting sort*, es necesario calcular el valor máximo, para poder guardar memoria del tamaño del máximo elemento, en base a ello hacer el *counting* de la línea 13, la línea 18 al 19 realiza los acumulados en el vector *count*, las líneas 20 a 22 realizan la reducción al vector de salida, tomando los elementos del vector de entrada y ubicándolos en el vector de salida según los valores de *counting* obtenidos en el paso anterior, este mismo proceso se realiza en C++, Python y Golang.

3.1.1. C++

```
1 int getMax(int array[], int size) {
2     int max = array[0];
3     for(int i = 1; i<size; i++) {
4         if(array[i] > max)
5             max = array[i];
6     }
7     return max;
8 }
9
10 void countSort(int array[], int size) {
11     int output[size+1];
12     int max = getMax(array, size);
13     int count[max];
14     for(int i = 0; i<=max; i++)
15         count[i] = 0;
16     for(int i = 0; i <size; i++)
17         count[array[i]]++;
18     for(int i = 1; i<=max; i++)
19         count[i] += count[i-1];
20     for(int i = size-1; i>=0; i--) {
21         output[count[array[i]]-1] = array[i];
22         count[array[i]] -= 1;
23     }
24     for(int i = 0; i<size; i++) {
25         array[i] = output[i];
26     }
27 }
```

Listing 2: CountingSort.h

3.1.2. Python

```
1 def countSort(arr):
2     output = [0 for i in range(len(arr))]
3     count = [0 for i in range(256)]
4     ans = [0 for i in arr]
5
6     for i in arr:
7         count[i] += 1
8
9     for i in range(256):
10        count[i] += count[i-1]
11
12    for i in range(len(arr)):
13        output[count[arr[i]]-1] = arr[i]
14        count[arr[i]] -= 1
15
16    for i in range(len(arr)):
17        ans[i] = output[i]
18    return ans
```

Listing 3: CountingSort.py

3.1.3. Golang

```
1 func findMax(arr []int) int {
2     var temp int
3
4     temp = arr[0]
5
6     for _, e := range arr {
7         if temp < e {
8             temp = e
9         }
10    }
11
12    return temp
13 }
14
15 func makeRange(min, max int) []int {
16     a := make([]int, max-min+1)
17     for i := range a {
18         a[i] = 0
19     }
20     return a
21 }
22
23 func countingSort(arr []int) []int {
24     counter := makeRange(0, findMax(arr))
25     // count
26     for _, e := range arr {
27         counter[e] += 1
28     }
29     for i := 1; i < len(counter); i++ {
```

```

30     counter[i] += counter[i-1]
31 }
32 res := make([]int, len(arr))
33
34 for i := 0 ; i < len(arr) ; i++ {
35     e := arr[i] // elem to add
36     t := counter[e] - 1 // target pos
37
38     res[t] = e
39     counter[e] = counter[e] - 1
40 }
41
42 return res
43 }

```

Listing 4: CountingSort.go

3.2. Quick Sort

Para el algoritmo *Quick sort* se realizo un sub proceso llamado partición, este mismo realiza el corte del arreglo origen y va dividiéndolo de manera recursiva, guardando los índices y devolviendo la nueva partición, en la linea 26 se calcula el punto de partición, para luego lanzar quick sort para los lados izquierdos y derecho, cada partición consiste en recorrer de principio a fin el segmento del arreglo ubicando a la derecha e izquierda los elementos mayores y menores, lo único que se realiza es el guardado del índice y su ubicación en el punto medio para partir en mitades izquierda y derecha.

3.2.1. C++

```

1 int Particion(int arreglo[], int inicio, int fin)
2 {
3     int pivote=arreglo[fin];
4     int i=inicio-1;
5
6     int auxiliar;
7
8     for(int j=inicio;j<fin;j++){
9         if(arreglo[j]<=pivote){
10             i=i+1;
11             auxiliar=arreglo[i];
12             arreglo[i]=arreglo[j];
13             arreglo[j]=auxiliar;
14         }
15     }
16     auxiliar=arreglo[i+1];
17     arreglo[i+1]=arreglo[fin];
18     arreglo[fin]=auxiliar;
19     return i+1;
20 }
21 }

```



```

22
23 void QuickSort(int arreglo[], int inicio, int fin)
24 {
25     if(inicio<fin){
26         int medio=Particion(arreglo,inicio,fin);
27         QuickSort(arreglo,inicio,medio-1);
28
29         QuickSort(arreglo,medio+1,fin);
30     }
31 }

```

Listing 5: QuickSort.h

3.2.2. Python

```

1 def partition(arr, low, high):
2     i = (low-1)
3     pivot = arr[high]
4     for j in range(low, high):
5         if arr[j] <= pivot:
6             i = i+1
7             arr[i], arr[j] = arr[j], arr[i]
8     arr[i+1], arr[high] = arr[high], arr[i+1]
9     return (i+1)
10
11 def quickSort(arr, low, high):
12     if len(arr) == 1:
13         return arr
14     if low < high:
15         pi = partition(arr, low, high)
16         quickSort(arr, low, pi-1)
17         quickSort(arr, pi+1, high)
18     return arr

```

Listing 6: QuickSort.py

3.2.3. Golang

```

1 func quickSort(arr []int, low, high int) []int {
2     if low < high {
3         var p int
4         arr, p = partition(arr, low, high)
5         arr = quickSort(arr, low, p-1)
6         arr = quickSort(arr, p+1, high)
7     }
8     return arr
9 }
10
11 func partition(arr []int, low, high int) ([]int, int) {
12     pivot := arr[high]
13     i := low
14     for j := low; j < high; j++ {

```

```

15     if arr[j] < pivot {
16         arr[i], arr[j] = arr[j], arr[i]
17         i++
18     }
19 }
20 arr[i], arr[high] = arr[high], arr[i]
21 return arr, i
22 }

```

Listing 7: QuickSort.go

3.3. Insertion Sort

Para el *insertion sort* se utiliza cada elemento del arreglo para poder compararlo con los elementos de la izquierda, por ello se comienza con la posición 1, dado que esta si tiene un elemento a la izquierda, de esta manera se va ubicando los elementos en el arreglo del lado izquierdo, este procedimiento se realiza en cada uno de los lenguajes de programación.

3.3.1. C++

```

1 void InsertionSort(int arreglo[], int longitud){
2     for(int j=1; j<longitud; j++){
3         int clave=arreglo[j];
4         int i=j-1;
5         while(i>=0 && arreglo[i]>clave){
6             arreglo[i+1]=arreglo[i];
7             i=i-1;
8         }
9         arreglo[i+1]=clave;
10    }
11 }

```

Listing 8: InsertionSort.h

3.3.2. Python

```

1 def InsertionSort(arreglo):
2     for i in range(1, len(arreglo)):
3         clave = arreglo[i]
4         j = i - 1
5         while j >= 0 and clave < arreglo[j]:
6             arreglo[j + 1] = arreglo[j]
7             j = j - 1
8         arreglo[j + 1] = clave

```

Listing 9: InsertionSort.py

3.3.3. Golang

```
1 func insertionSort(arreglo []int) []int {
2     len := len(arreglo)
3     for i := 1; i < len; i++ {
4         for j := 0; j < i; j++ {
5             if arreglo[j] > arreglo[i] {
6                 arreglo[j], arreglo[i] = arreglo[i], arreglo[j]
7             }
8         }
9     }
10    return arreglo;
11 }
```

Listing 10: InsertionSort.go

3.4. *Heap Sort*

El proceso del *Heap sort* consiste en aplicar un proceso de amontonar o *heapify*, el cual ordena el sub árbol, habiendo obtenido el máximo o mínimo, se separa este del resto del árbol, ese proceso se realiza en la línea 20, luego se realiza el proceso una vez mas en la línea 24, antes de haber movido el elemento al final del arreglo.

3.4.1. C++

```
1 void amontonar(int arr[], int n, int i)
2 {
3     int masGrande = i;
4     int izquierda = 2*i + 1;
5     int derecha = 2*i + 2;
6     if (izquierda < n && arr[izquierda] > arr[masGrande])
7         masGrande = izquierda;
8     if (derecha < n && arr[derecha] > arr[masGrande])
9         masGrande = derecha;
10    if (masGrande != i)
11    {
12        swap(arr[i], arr[masGrande]);
13        amontonar(arr, n, masGrande);
14    }
15 }
16
17 void heapSort(int arr[], int n)
18 {
19     for (int i = n / 2 - 1; i >= 0; i--)
20         amontonar(arr, n, i);
21     for (int i=n-1; i>0; i--)
22     {
23         swap(arr[0], arr[i]);
24         amontonar(arr, i, 0);
25     }
```

26 }

Listing 11: HeapSort.txt

3.4.2. Python

```
1
2 def heapify(unsorted, index, heap_size):
3     largest = index
4     left_index = 2 * index + 1
5     right_index = 2 * index + 2
6     if left_index < heap_size and unsorted[left_index] > unsorted[
largest]:
7         largest = left_index
8
9     if right_index < heap_size and unsorted[right_index] > unsorted[
largest]:
10        largest = right_index
11
12    if largest != index:
13        unsorted[largest], unsorted[index] = unsorted[index], unsorted[
largest]
14        heapify(unsorted, largest, heap_size)
15
16 def HeapSort(unsorted):
17     n = len(unsorted)
18     for i in range(n // 2 - 1, -1, -1):
19         heapify(unsorted, i, n)
20     for i in range(n - 1, 0, -1):
21         unsorted[0], unsorted[i] = unsorted[i], unsorted[0]
22         heapify(unsorted, 0, i)
23     return unsorted
```

Listing 12: Heapsort.py

3.4.3. Golang

```
1
2 func heapSort(A []int) []int {
3     buildMaxHeap(A)
4     size := len(A)
5
6     for size > 1 {
7         A[0], A[size-1] = A[size-1], A[0]
8         size -= 1
9
10        siftDown(A[:size], 0)
11    }
12    return A
13 }
14
15 func buildMaxHeap(A []int) {
```

```

16  for i := (len(A) / 2) - 1; i >= 0; i -= 1 {
17      siftDown(A, i)
18  }
19 }
20
21 func siftDown(A []int, i int) {
22     largest := i
23     l, r := getChildIndices(i)
24     if l < len(A) && A[l] > A[i] {
25         largest = l
26     }
27     if r < len(A) && A[r] > A[largest] {
28         largest = r
29     }
30     if largest != i {
31         A[i], A[largest] = A[largest], A[i]
32         siftDown(A, largest)
33     }
34 }
35
36 func getParentIndex(childIndex int) int {
37     if childIndex % 2 == 0 {
38         return (childIndex - 2) / 2
39     } else {
40         return (childIndex - 1) / 2
41     }
42 }
43
44 func getChildIndices(parentIndex int) (int, int) {
45     return (2 * parentIndex + 1), (2 * parentIndex + 2)
46 }

```

Listing 13: Heapsort.go

4. Resultados

Para el reporte de resultados se han usado gráficos variados, dentro de los cuales tenemos dos tipos:

1. Comparaciones de Algoritmos: En estos gráficos se ha comparados los 4 algoritmos en 1 sola gráfica, para poder identificar el mas rápido.
2. Tiempo Promedio Límites Superiores e Inferiores, para el estudio del límite superior, caso promedio y límite superior, se ha considerado como promedio el promedio de los 5 casos, el límite superior se ha considerado como la media mas 2 desviaciones estándar, el límite inferior es la media menos 2 desviaciones estándar, esto se realizo para cada uno de los algoritmos.

4.1. Comparación con Python, C++ y Golang

4.1.1. Comparación de *Counting Sort*

Gráfica del Algoritmos en Python, C++ y Golang.

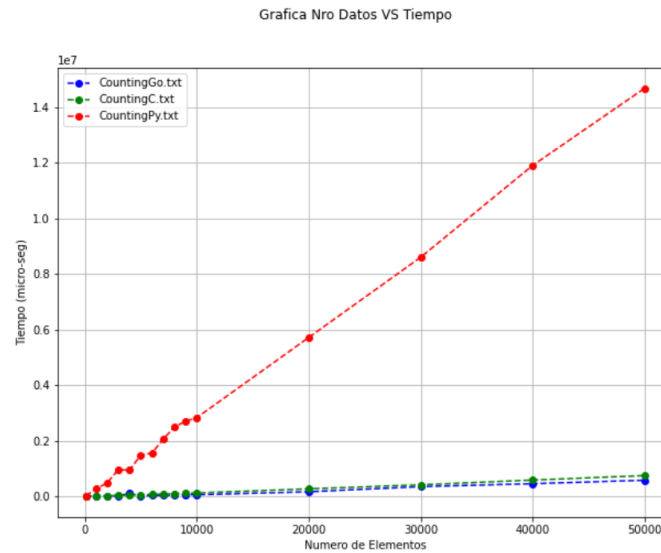


Figura 7: Comparación de *Counting Sort*

4.1.2. Comparación de *Quick Sort*

Comparación del Algoritmo en Python, C++ y Golang.

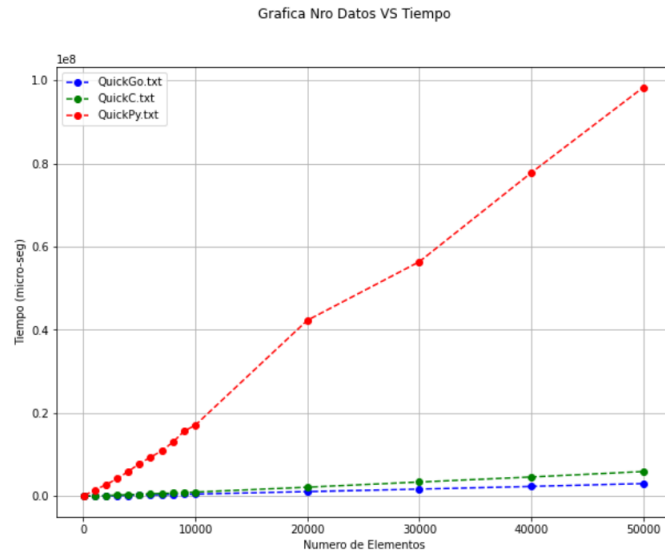


Figura 8: Comparación de *Quick Sort*

4.1.3. Comparación de *Insertion Sort*

Comparación del Algoritmo en Python, C++ y Golang.

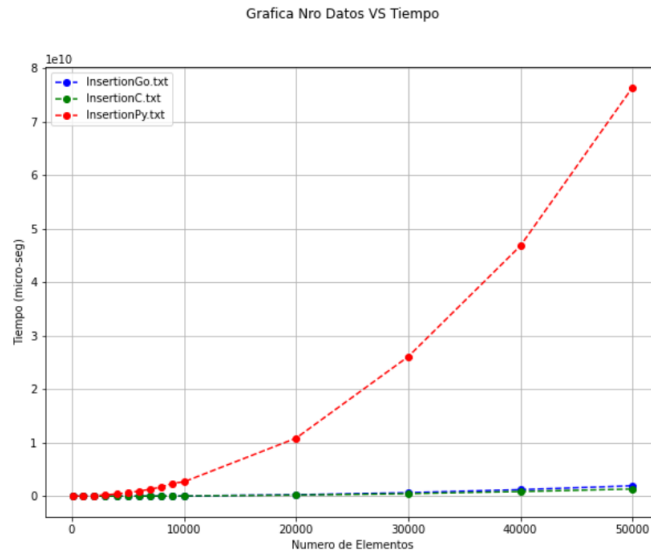


Figura 9: Comparación de *Insertion Sort*

4.1.4. Comparación *Heap Sort*

Comparación del Algoritmo en Python, C++ y Golang.

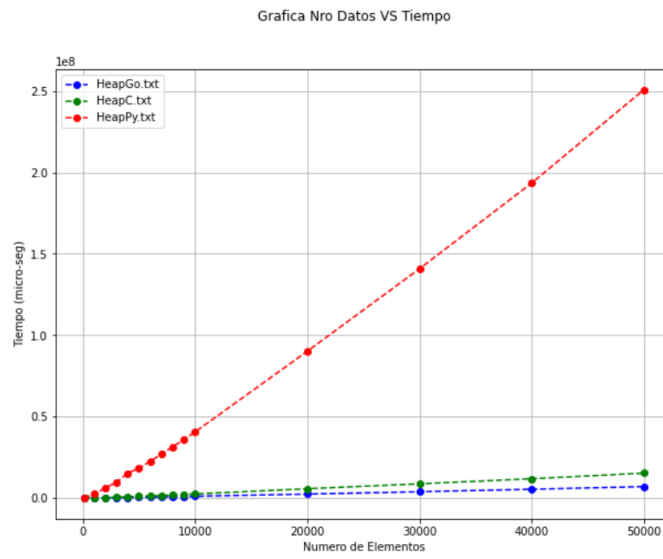


Figura 10: Comparación de *Heap Sort*

4.2. Comparación con C++

4.2.1. Comparación de Algoritmos en C++

Gráfica del Algoritmos en Python, C++ y Golang.

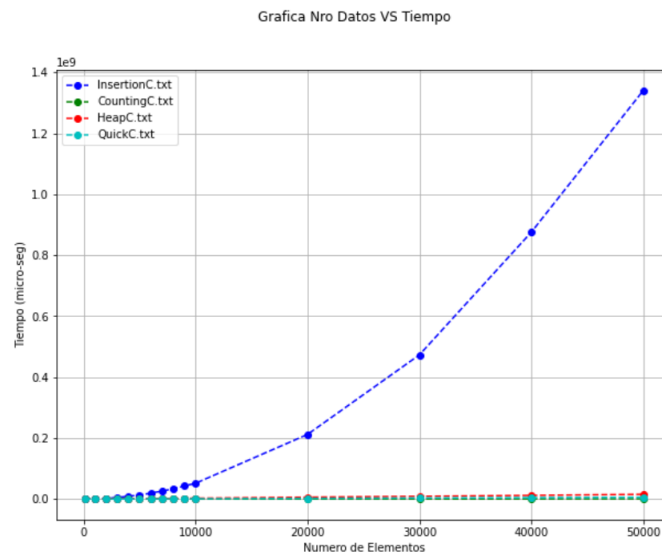


Figura 11: Comparación de Algoritmos en C++

4.2.2. *Insertion Sort* en C++

Comparación del Algoritmo en Python, C++ y Golang.

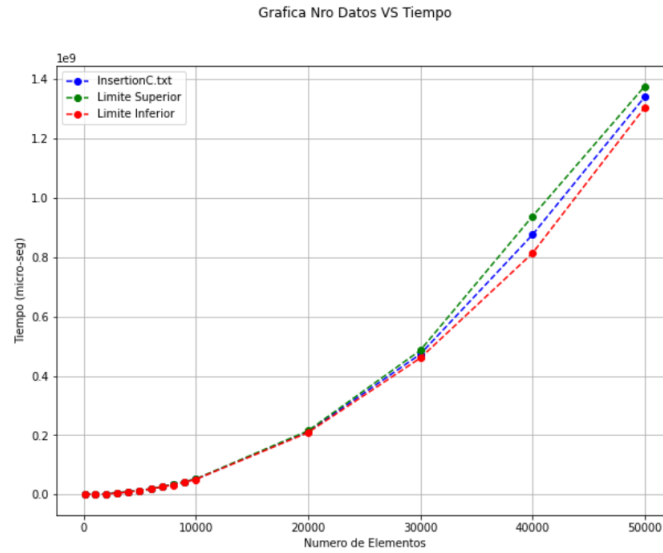


Figura 12: Comparación de *Insertion Sort*

4.2.3. *Counting Sort* en C++

Comparación del Algoritmo en Python, C++ y Golang.

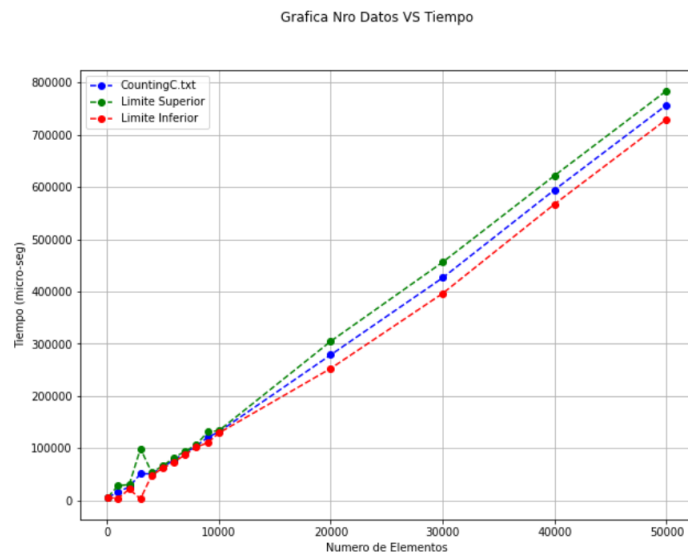


Figura 13: Comparación de *Counting Sort*

4.2.4. *Heap Sort* en C++

Comparación del Algoritmo en Python, C++ y Golang.

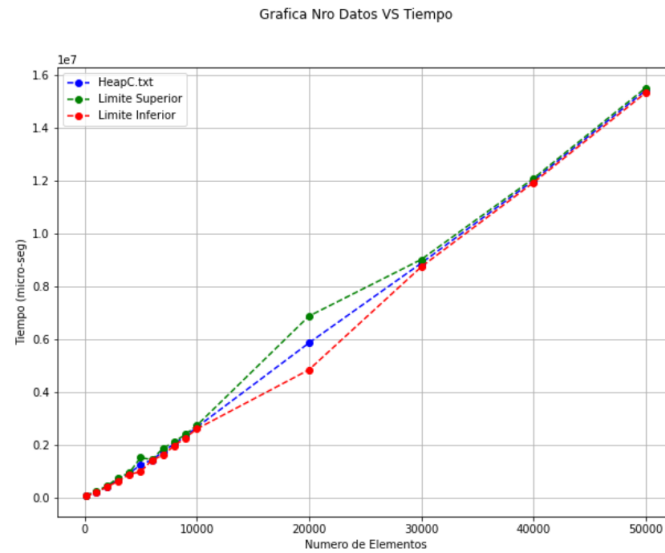


Figura 14: Comparación de *Heap Sort*

4.2.5. *Quick Quick Sort* en C++

Comparación del Algoritmo en Python, C++ y Golang.

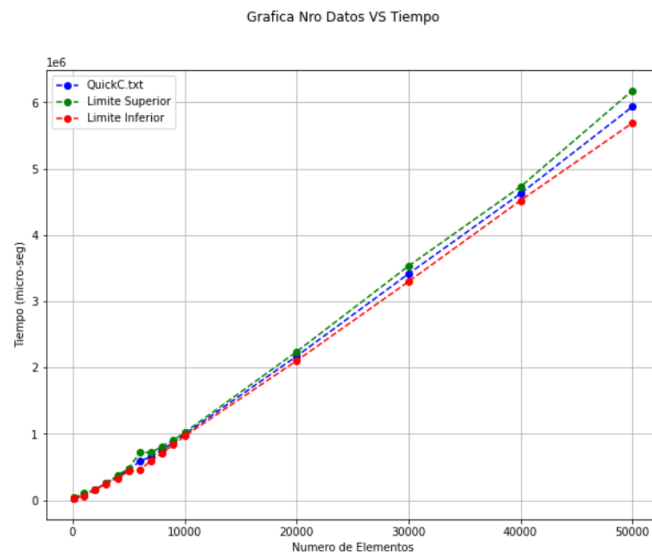


Figura 15: Comparación de *Quick Quick Sort*

4.3. Comparación con Golang

4.3.1. Comparación de Algoritmos en Golang

Gráfica del Algoritmos en Python, C++ y Golang.

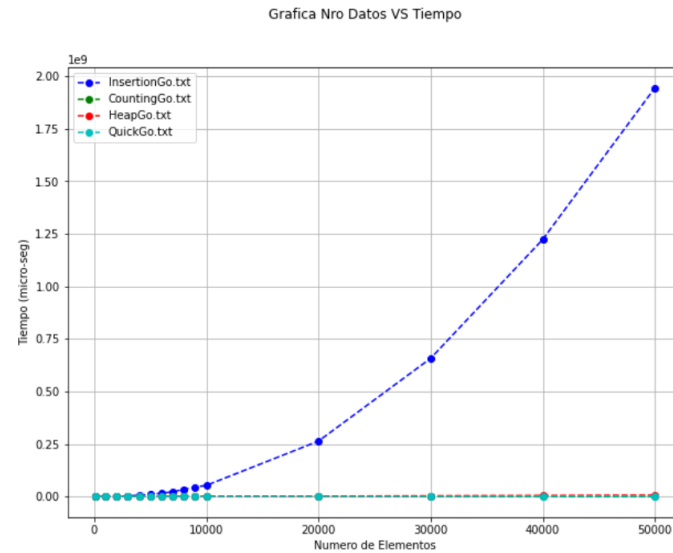


Figura 16: Comparación de Algoritmos en Golang

4.3.2. *Insertion Sort* en Golang

Comparación del Algoritmo en Python, C++ y Golang.

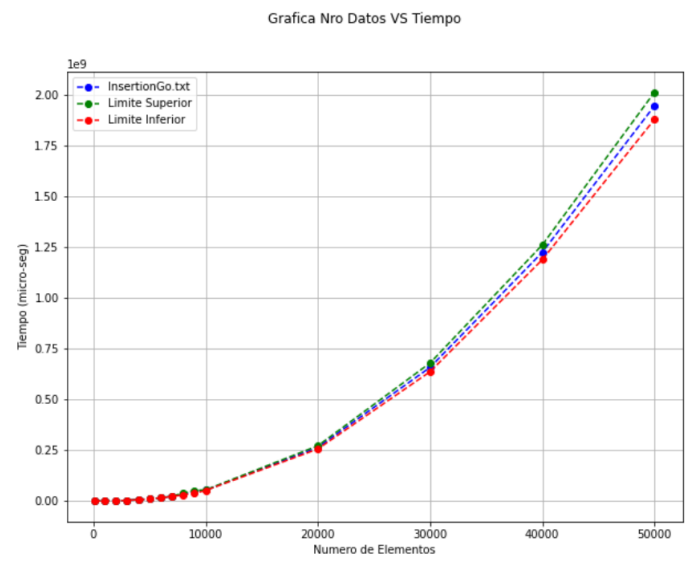


Figura 17: *Insertion Sort* en Golang

4.3.3. *Counting Sort* en Golang

Comparación del Algoritmo en Python, C++ y Golang.

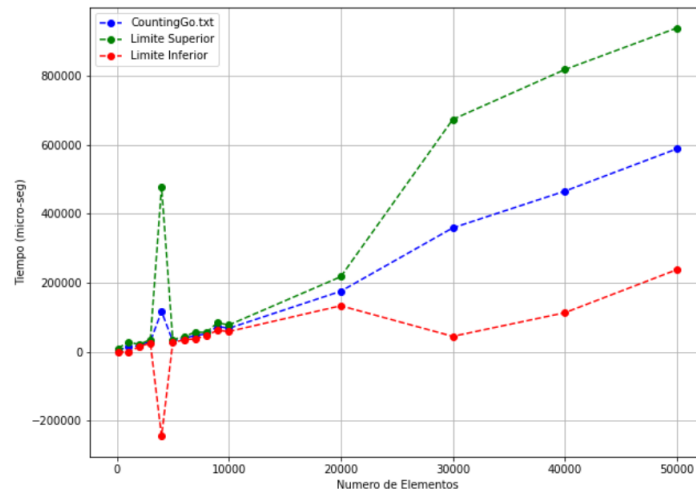


Figura 18: *Counting Sort* en Golang

4.3.4. *Heap Sort* en Golang

Comparación del Algoritmo en Python, C++ y Golang.

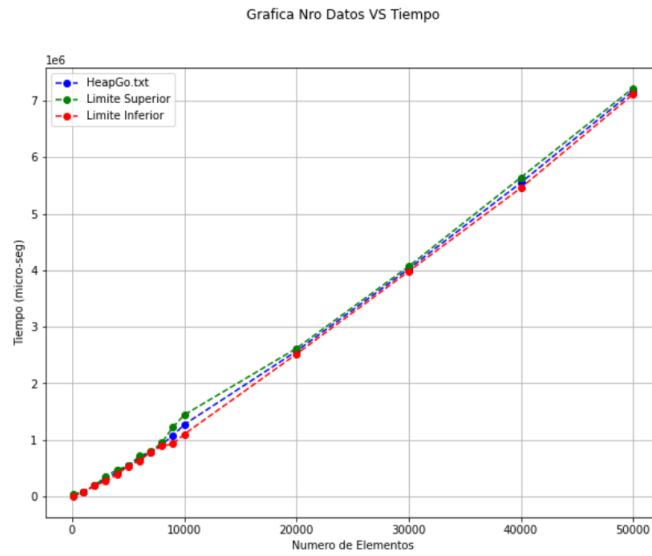


Figura 19: *Heap Sort* en Golang

4.3.5. *Quick Sort* en Golang

Comparación del Algoritmo en Python, C++ y Golang.

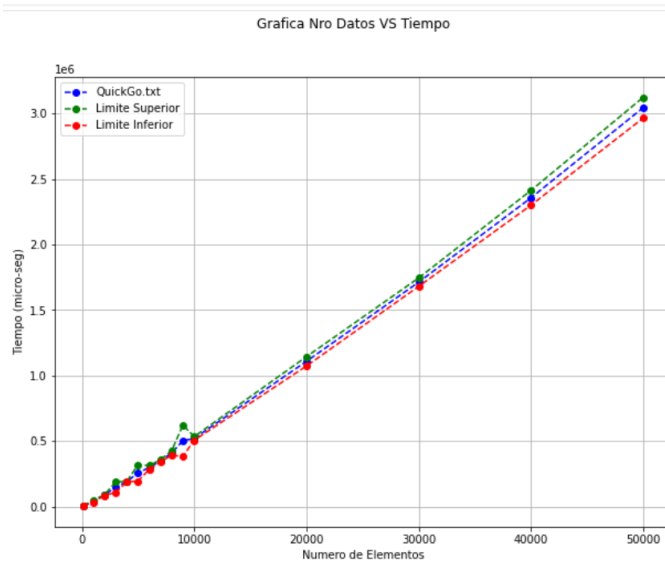


Figura 20: *Quick Sort* en Golang

4.4. Comparación con Python

4.4.1. Comparación de Algoritmos en Python

Gráfica del Algoritmos en Python, C++ y Golang.

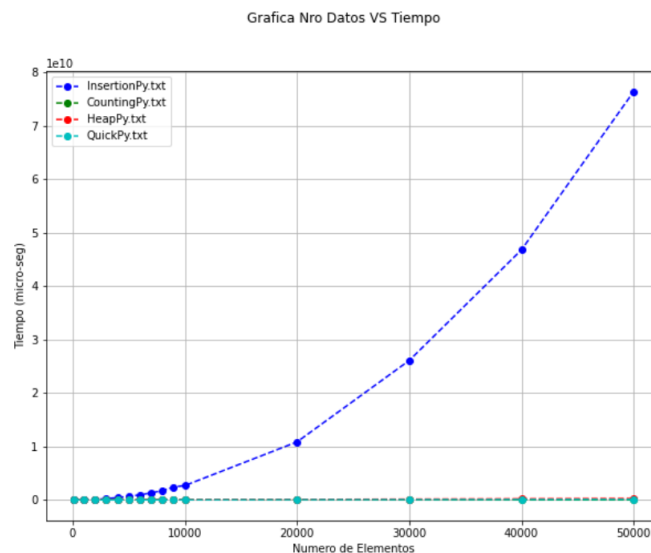


Figura 21: Comparación de Algoritmos en Python

4.4.2. *Insertion Sort* en Python

Comparación del Algoritmo en Python, C++ y Golang.

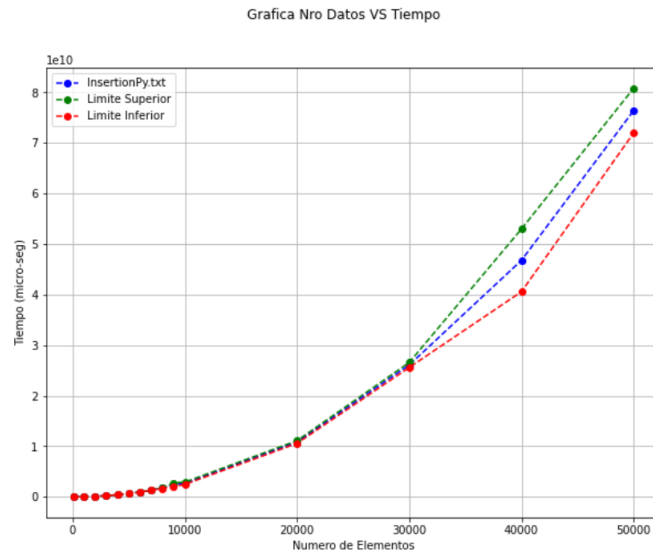


Figura 22: *Insertion Sort* en Python

4.4.3. *Counting Sort* en Python

Comparación del Algoritmo en Python, C++ y Golang.

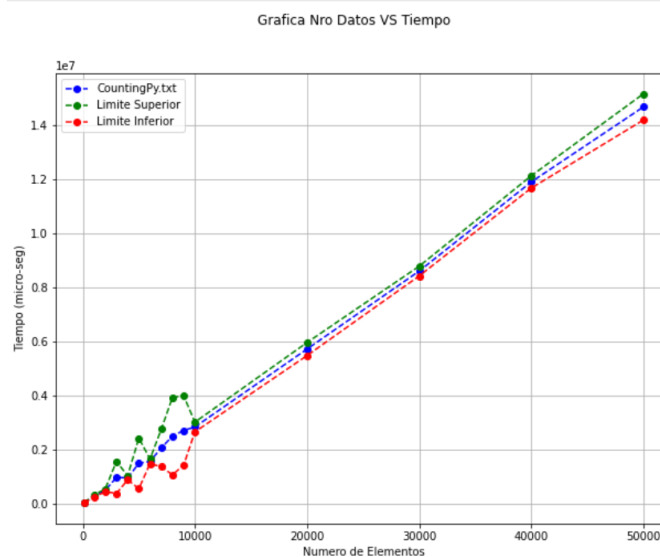


Figura 23: *Counting Sort* en Python

4.4.4. *Heap Sort* en Python

Comparación del Algoritmo en Python, C++ y Golang.

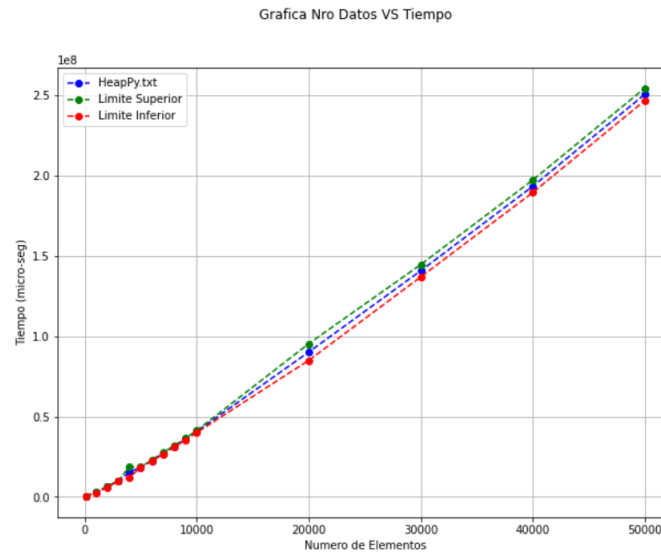


Figura 24: *Heap Sort* en Python

4.4.5. *Quick Sort* en Python

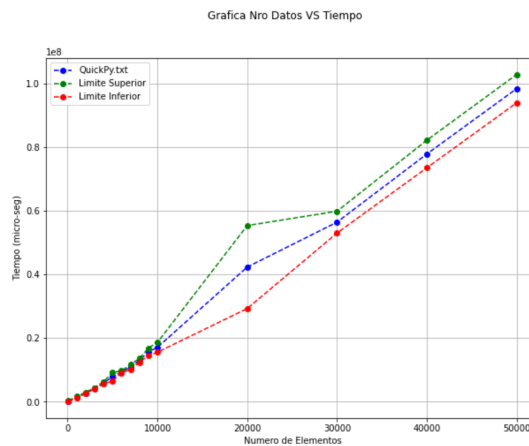


Figura 25: *Quick Sort* en Python

4.5. Tablas Comparativa de Tiempos

Los tiempos que se muestran en las tablas son el promedio de 5 muestras con cada tamaños de entrada formateadas en nanosegundos.

4.5.1. Tiempos de ejecución: *Counting Sort*

Tamaño	Lenguaje Golang	Lenguaje C++	Lenguaje Python
100	4156.4	6411.2	28610.2
1000	13108.6	16571.4	277566.9
2000	17744.0	26460.4	493144.9
3000	29865.4	51471.6	968122.4
4000	116422.8	51320.6	954723.3
5000	30517.2	64278.6	1488065.7
6000	38804.8	78167.0	1567077.6
7000	46604.0	91283.8	2057838.4
8000	51361.0	104795.4	2499723.4
9000	73105.8	121225.4	2708625.7
10000	67685.6	131594.4	2835369.1
20000	175157.4	278895.0	5719041.8
30000	358891.8	426290.2	8603239.0
40000	465343.2	594650.0	11903095.2
50000	588081.2	756761.2	14678049.0

Cuadro 5: Tiempos de ejecución del algoritmo *Counting Sort*

4.5.2. Tiempos de ejecución: *Quick Sort*

Tamaño	Lenguaje Golang	Lenguaje C++	Lenguaje Python
100	3393.8	29545.6	146007.5
1000	41710.2	80245.0	1518249.5
2000	86745.4	160182.8	2728891.3
3000	147516.6	255427.0	4216384.8
4000	190250.4	353236.2	5927562.7
5000	256520.0	458179.0	7825803.7
6000	298074.8	586476.2	9375333.7
7000	349201.4	658908.2	10919904.7
8000	409211.6	762335.6	13009119.0
9000	503380.0	871668.4	15661668.7
10000	519730.0	993605.2	17081308.3
20000	1107891.6	2171292.4	42290687.5
30000	1712465.8	3415140.6	56382703.7
40000	2355919.2	4624127.8	77776670.4
50000	3043727.2	5934884.6	98295640.9

Cuadro 6: Tiempos de ejecución del algoritmo *Quick Sort*

4.5.3. Tiempos de ejecución: *Insertion Sort*

Tamaño	Lenguaje Golang	Lenguaje C++	Lenguaje Python
100	34279.6	9664.8	238466.2
1000	1047769.2	560293.2	24469757.0
2000	1618634.6	2146801.2	101745653.1
3000	3545198.0	4863337.4	231694602.9
4000	6452998.0	8871870.6	419051647.1
5000	10572702.4	13384660.0	648374462.1
6000	15932791.8	19018828.8	952532958.9
7000	23171703.6	26055904.2	1307987546.9
8000	34123140.8	33577623.6	1729411411.2
9000	44976422.2	42766146.8	2414094448.0
10000	53407743.2	51611769.6	2681028413.7
20000	264132010.6	211818186.2	10851650571.8
30000	657936185.8	473043358.2	26091095590.5
40000	1223976990.6	875843928.2	46787051725.3
50000	1945301778.2	1340750614.6	76417606306.0

Cuadro 7: Tiempos de ejecución del algoritmo *Insertion Sort*

4.5.4. Tiempos de ejecución: *Heap Sort*

Tamaño	Lenguaje Golang	Lenguaje C++	Lenguaje Python
100	17632.2	55352.4	199508.6
1000	87206.6	195362.0	2769327.1
2000	189115.2	412048.0	6170511.2
3000	315177.6	672130.4	9947490.6
4000	431417.6	904263.6	15331888.1
5000	539877.0	1239145.0	18486928.9
6000	673315.6	1413743.6	22559595.1
7000	787593.2	1730729.4	27085971.8
8000	922658.4	2028979.8	31501817.7
9000	1079712.4	2321449.8	35928535.4
10000	1272634.2	2656174.8	40650987.6
20000	2569350.4	5853268.2	90158414.8
30000	4024258.2	8876363.8	140787458.4
40000	5548282.2	12012330.8	193399429.3
50000	7164878.0	15432487.0	250664281.8

Cuadro 8: Tiempos de ejecución del algoritmo *Heap Sort*

5. Conclusiones

1. La implementación de todos los algoritmos en el lenguaje de C++ tiene mejores resultados en comparación con los algoritmos de Python y Golang.
2. Los gráfico del tiempo de procesamiento de cada uno de los algoritmos se rigen en función matemática que representa su complejidad algorítmica.
3. El tiempo de procesamiento va depender mucho del tamaño de data que ingresemos.
4. Las características propias del lenguaje influyen directamente en el tiempo de respuesta del algoritmo.